Embedded Linux Conference
North America

- Tracing resource-constrained embedded systems using eBPF

Adrian Ratiu, Collabora Ltd

THE LINUX FOUNDATION

Agenda

- About me
- Embedded / IoT woes
- How does eBPF fit in?
- Quick eBPF / BCC introduction, benefits
- Approaches to eBPF on embedded devices
- Trade-offs, specific projects pros/cons
- Ways forwards

About me.

I enjoy

working in a company of awesome FOSS-oriented people at Collabora

work with companies who “get it” when using FOSS

work to help companies “get it” and be successful

I also really enjoy

Taking systems apart and modifying them

Projects like OpenEmbedded/Yocto, Buildroot/OpenWRT

Always looking for new tech to improve development and debugging of embedded devices

Learning about eBPF (just a user, not an expert)

A strong dislike of locked-down devices /
that lock owner usage without very good reasons

Embedded and the IoT

- “Smart” devices everywhere
- Increasingly powerful, complex, connected hardware
- Much more capable than default software installations allow
- Software complexity is also rising

    (embedded systems now programmed in JavaScript)
- Obvious privacy, security and vendor lock-in concerns

Embedded problems

Devices have more power and run modern software

yet they are really hard to

develop, debug, maintain and extend

Embedded problems

Why?

Embedded problems

Why?

Increased SW/HW complexity

+

Embedded-specific **resource constraints**

Resource constraints

- Enough memory to run just a specific pre-built workload
- Cross-compiling and flashing/provisioning
- Special “Embedded Linux” distributions
- RT deadline requirements
- Ergonomics trade-offs, lack of HW ports
- Licensing requirements (no GPLv3...)
- Weird HW combinations, countless HW revisions
- Throw-away HW, planned obsolescence
- Low quality Out-Of-Tree drivers
- <Add your own pet-peeve here>

Creative solutions against constraints

- Debug symbol servers and remote GDB sessions
- Booting rootfs over the network
- Special protocols for diagnostics/log/trace
- Debug vs Release images, “developer mode”
- And so on

Creative solutions against constraints

- Debug symbol servers and remote GDB sessions
- Booting rootfs over the network
- Special protocols for diagnostics/log/trace
- Debug vs Release images, “developer mode”
- And so on

Here comes eBPF

Wait a minute

Embedded-eBPF sounds like a solution in search of a problem...

Wait a minute

Embedded-eBPF sounds like a solution in search of a problem…

It kind of is.

“Embedded” engineers drooling over tools of “Cloud” engineers

Would like to have same system observability powers

Precedent: SMP on embedded

Explaining eBPF / BCC in a few slides!

BCC automates

VM bytecode

Kernel ↥ Userspace

Links at the end for better learning resources.

VM running bytecode in the Linux kernel

Bytecode loaded from userspace via bpf() syscall

Verified for safety, unsafe => syscall rejects bytecode

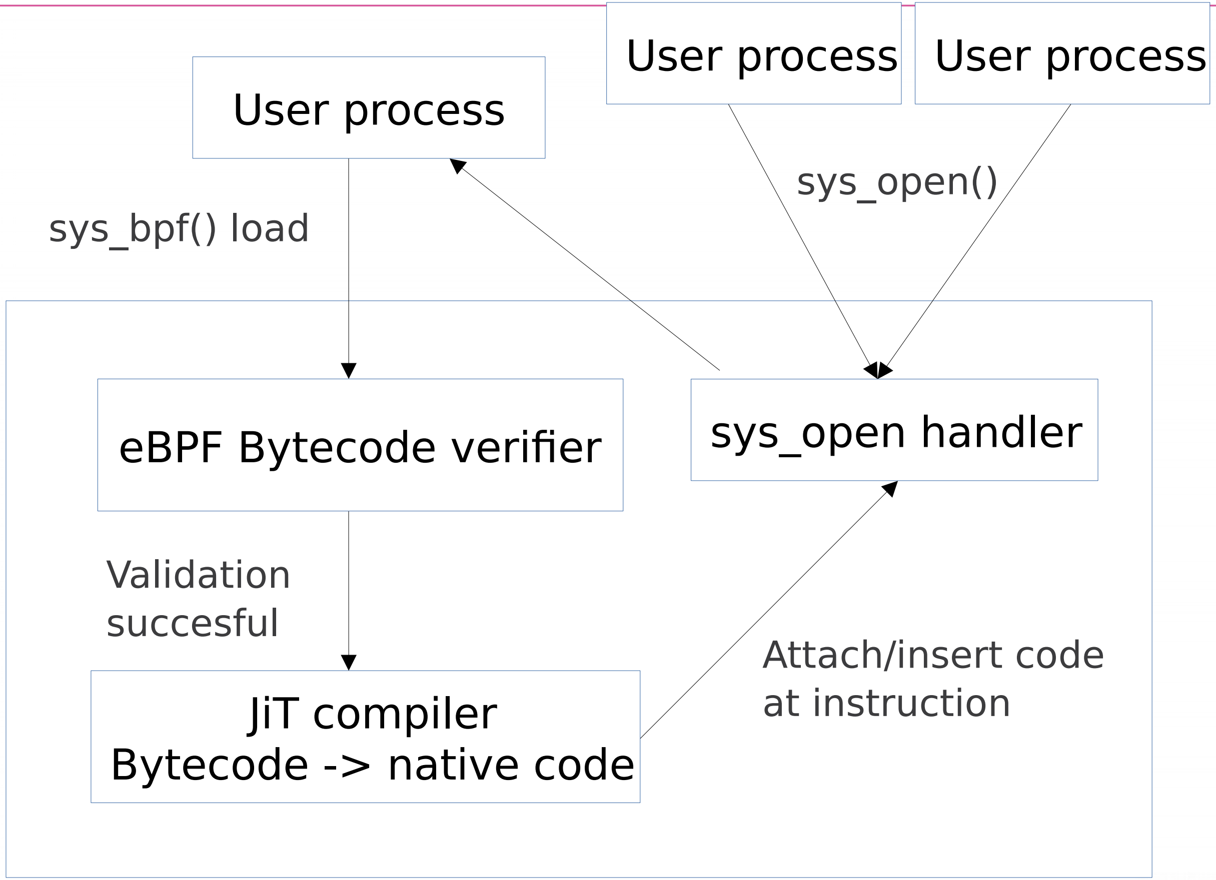
Bytecode compiled to native machine code

Native code inserted in execution paths

Event-driven programming

Native code runs and collects data

Data shared with userspace



User process
User process
User process
sys_open()
sys_bpf() load
eBPF Bytecode verifier
sys_open handler
Validation
succesful
Attach/insert code
at instruction
JiT compiler
Bytecode -> native code

How does userspace produce that bytecode?

```
0:  79 12 60 00 00 00 00 00    r2 = *(u64 *)(r1 + 96)
1:  7b 2a 98 ff 00 00 00 00    *(u64 *)(r10 - 104) = r2
2:  79 17 70 00 00 00 00 00    r7 = *(u64 *)(r1 + 112)
3:  85 00 00 00 0e 00 00 00    call 14
4:  bf 06 00 00 00 00 00 00    r6 = r0
5:  b7 09 00 00 00 00 00 00    r9 = 0
6:  7b 9a c0 ff 00 00 00 00    *(u64 *)(r10 - 64) = r9
7:  bf 73 00 00 00 00 00 00    r3 = r7
8:  07 03 00 00 18 00 00 00    r3 += 24
9:  bf a1 00 00 00 00 00 00    r1 = r10
11: 07 01 00 00 c0 ff ff ff    r1 += -64
12: b7 02 00 00 08 00 00 00    r2 = 8
13: 85 00 00 00 04 00 00 00    call 4
```

How does userspace produce that bytecode?

Directly write it byte by byte! 😱

```
0:   79 12 60 00 00 00 00 00    r2 = *(u64 *)(r1 + 96)
1:   7b 2a 98 ff 00 00 00 00    *(u64 *)(r10 - 104) = r2
2:   79 17 70 00 00 00 00 00    r7 = *(u64 *)(r1 + 112)
3:   85 00 00 00 0e 00 00 00    call 14
4:   bf 06 00 00 00 00 00 00    r6 = r0
5:   b7 09 00 00 00 00 00 00    r9 = 0
6:   7b 9a c0 ff 00 00 00 00    *(u64 *)(r10 - 64) = r9
7:   bf 73 00 00 00 00 00 00    r3 = r7
8:   07 03 00 00 18 00 00 00    r3 += 24
9:   bf a1 00 00 00 00 00 00    r1 = r10
11:  07 01 00 00 c0 ff ff ff    r1 += -64
12:  b7 02 00 00 08 00 00 00    r2 = 8
13:  85 00 00 00 04 00 00 00    call 4
```

Clang can translate “restricted C” into eBPF bytecode

Much easier than assembling bytes like the 1960s

Still hard to write userspace interaction

Clang can translate “restricted C” into eBPF bytecode

Much easier than assembling bytes like the 1960s

Still hard to write userspace interaction

BCC: the **B**PF **C**ompiler **C**olection

- Framework to ease writing userspace eBPF programs
- Abstracts Clang and sys_bpf() interaction
- “restricted C” compiled & loaded in kernel on-the-fly
- Provides Python, Lua and Go bindings
- Provides production ready BCC-tools

BCC program

```
#!/usr/bin/env python
from bcc import BPF

csrc = """
#include <uapi/linux/ptrace.h>

int kprobe__do_sys_open(struct pt_regs *ctx)
{
        char file_name[256];
        bpf_probe_read(&file_name, sizeof(file_name), PT_REGS_PARM1(ctx));
        bpf_trace_printk(fmt, sizeof(fmt), file_name);
}
"""

b = BPF(text=csrc)
b.attach_kprobe(event="do_sys_open", fn_name="kprobe__do_sys_open")
while True:
    time.sleep(1)
```

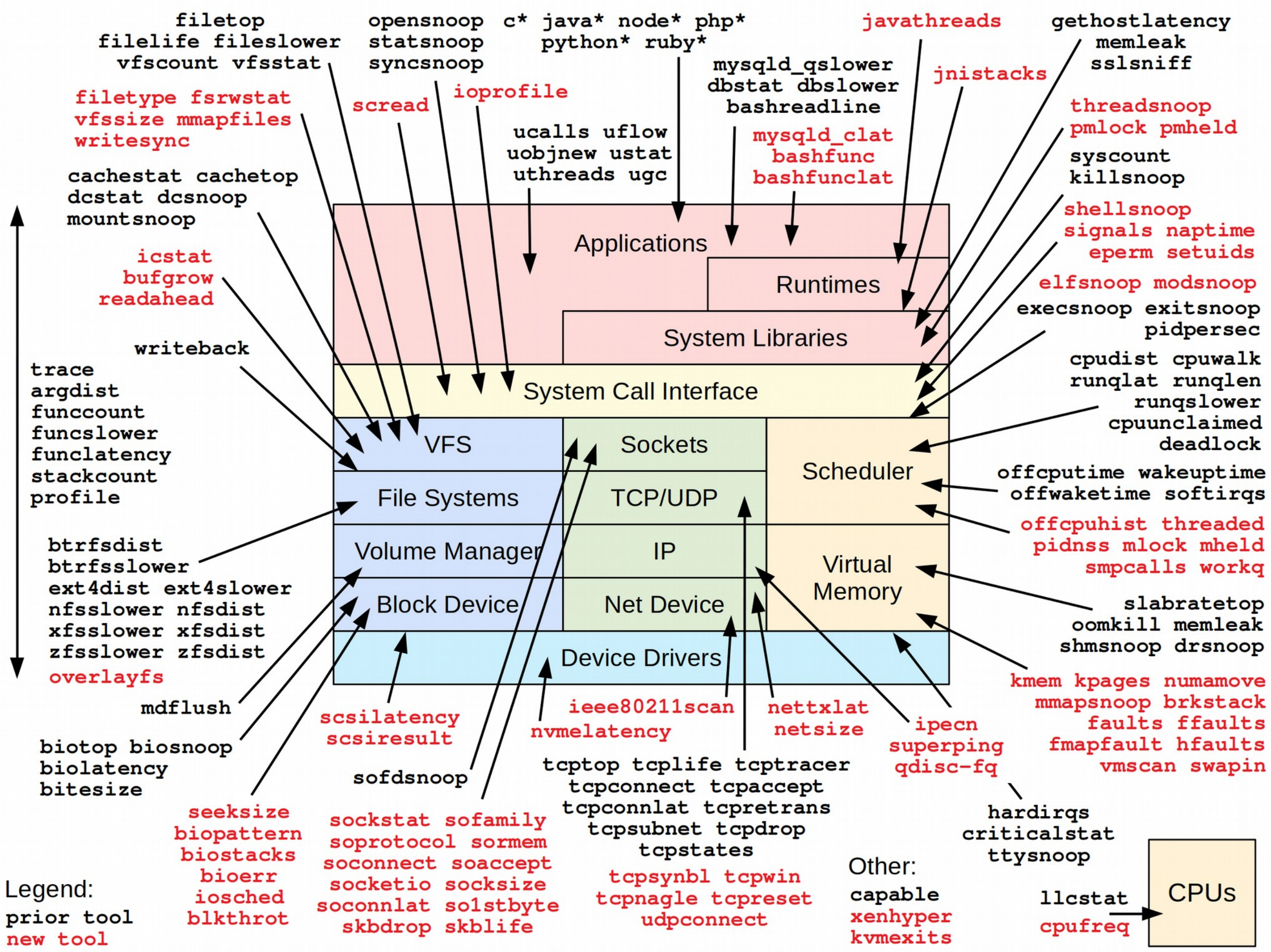
BCC program

```
#!/usr/bin/env python
from bcc import BPF

csrc = """
#include <uapi/linux/ptrace.h>

int kprobe__do_sys_open(struct pt_regs *ctx)
{
        char file_name[256];
        bpf_probe_read(&file_name, sizeof(file_name), PT_REGS_PARM1(ctx));
        bpf_trace_printk(fmt, sizeof(fmt), file_name);
}
"""

b = BPF(text=csrc)
b.attach_kprobe(event="do_sys_open", fn_name="kprobe__do_sys_open")
while True:
    time.sleep(1)
```

Compiled to bytecode
Loaded & runs in kernel
Collects data
Sends to userspace

Calls Clang to compile above code
Loads bytecode via bpf()

New tools developed for the book **BPF Performance Tools: Linux System and Application Observability** by Brendan Gregg (Addison Wesley, 2019), which also covers **prior BPF tools**



Real power comes with the **BCC tools**

Executive summary eBPF benefits

- System-wide observability
- No crashes / hangs
- No performance degradations
- Real-time production workload analysis
- Can be always enabled (no special debug builds) *
- Fully upstream kernel feature, active community
- Big collection of production-ready tools
- More than just observing a system

Packet filtering, hw offloading

Executive summary eBPF benefits

Convincing, yes?

- System-wide observability
- No crashes / hangs
- No performance degradations
- Real-time production workload analysis
- Can be always enabled (no special debug builds)
- Fully upstream kernel feature, active community
- Big collection of production-ready tools
- More than just observing a system

  Packet filtering, hw offloading

eBPF meets embedded

general / embedded-specific problems

multiple approaches

project advantages / disadvantages

trade-offs, no silver bullet

General problem: portability / cross-compilation

Poking “outside” from within the eBPF VM

- VM has generic 64 bit instructions/registers/pointers
- Difficulty accessing 32 bit kernel/user data structures
- VM is capable of 32 bit register subaddressing
- Pointer arithmetic hacks can access 32bit offset data

  Very fragile, not portable

- Better solution: **B**PF **T**ype **F**ormat adds type info to compiled eBPF

  (part of **C.O.R.E.**)

General problem: portability / cross-compilation

Portable eBPF (**C**ompile **O**nce, **R**un **E**verywhere)

- Dream: run precompiled eBPF an any machine and expect it to work
- Slimmer version of BCC using BTF info, no Clang runtime compilation
  (structure offsets built in BTF sections, macro identifiers → BPF variables)
- Current runtime compilation uses version/config specific C headers
  - Backwards, not forwards compatible
  - Manually copying non-UAPI structures to “restricted C”
  - Big variation of Linux kernel configs → *header* structures
- Kernel >= 5.2 can remove header filesystem dependency (kinda unrelated)
- Work on-going

THE LINUX FOUNDATION

General problem: Security and unpriviledged eBPF

Running eBPF programs requires root / CAP_SYS_ADMIN

- eBPF code is assumed not malicious
- CAP_BPF will be added to restrict attack surface
- Unpriviliged eBPF unlikely to happen

Care must be taken when running eBPF code in production

- Don’t run arbitrary eBPF supplied by untrusted users
- Use additional security mechanisms like verified boot

Awesome (as always) relevant LWN.net article and comments:

https://lwn.net/Articles/796328/

Approach 1: Precompiled eBPF + custom userspace

PRO:
Lightest footprint possible
(few kb C program)

Kernel provides helper libbpf
(useful starting point)

CON:
Need to write from scartch
Userspace sys_bpf() interaction

Can get complex, hard to maintain
No pre-existing community

Some examples provided by Linux kernel tree
in samples/bpf/

Approach 2: Use BCC directly

PRO:
Vanilla upstream BCC
Full framework capabilities
All BCC-tools available
Well tested, good performance

CON:
Installs and links against Clang
Depends on Python (bcc-tools)
~ 300 MB storage

Will benefit from C.O.R.E., but will still require python

Example project: Androdeb
(Requires > 2GB storage)

Approach 3: BPFd

```
+---------------------------------+    SSH, Telnet     +------------------------------+
| Restricted C -> Python -> BCC <-=-------------------=-> BPFd <-> libbpf <-> kernel |
|                            |    |                    +------------------------------+
|   device kernel source -> LLVM  |                                   Embedded device
+---------------------------------+
                  Host machine
```

Project abandoned due to high maintenance cost

PRO:
100 kb bin + libc dependency
Full framework capabilities
All BCC-tools available

CON:
Hard to maintain BCC<>BPFd interaction
Host + target + transport
dependent architecture

Approach 4: DSL compiler from scratch - Ply

```
+-----------------------------------+    ssh/nfs/http    +----------------------------+
|            ply source -+--> GCC -=---------------------=-> ply compiler <-> kernel |
|                        |          |                    |    |                      |
| device kernel source -+           |                    |  script                   |
+-----------------------------------+                    +----------------------------+
               Developer machine                                   Embedded device
```

```
ply 'kprobe:i2c_transfer { print(stack); }'
```

PRO:
50 kb bin + libc dependency
High level, AWK-inspired DSL
Self-contained
Easy to build & deploy

CON:
Lack of kernel/user interaction control
Lack of BCC-tools diversity
Under heavy development
Ply binary is not portable

Approach 5: Replace BCC Python userspace with Go

PRO:

~2 mb static-compiled eBPF loader

Full control over kernel/user interaction

Good coverage of BCC API bindings

CON:

BCC-tools need rewriting in Go :)

Not much documentation

```
+-----------------------------------+   ssh/nfs/http   +-----------------------------+
|          Restricted C -+--> LLVM -=------------------=-> eBPF_ELF.o               |
|                        |          |                   |      |                      |
| device kernel source -+          |                   |      | loaded in kernel     |
|                                  |                   |      |                      |
|            Go source -> go build -=------------------=-> go_loader.bin <-> kernel |
+-----------------------------------+                  +-----------------------------+
                    Developer machine                                  Embedded device
```

Full execsnoop reimplementation:

https://github.com/iovisor/gobpf/blob/master/examples/bcc/execsnoop/execsnoop.go

Ways forward

- C.O.R.E. needs to be as succesful as possible
- With C.O.R.E. BCC will be more lightweight
- Gobpf can eliminate the Python dependency
- BPFd reached a dead end
- Ply is standalone, will continue its awesomeness
- eBPF on embedded is already useful today<
- Much work remaining

Recommended learning resources:

- LWN.net eBPF articles https://lwn.net/
- Brendan Gregg’s blog: http://www.brendangregg.com/blog/
- BPF Performance Tools: Linux System and Application Observability, by Brendan Gregg, published by Addison Wesley (2019)
- Collabora eBPF blog posts

  https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/
- Internet Search has wealth of information on eBPF

Thank you!

Psst...
We're hiring!

Embedded Linux Conference

North America

Embedded Linux Conference

North America